FUSE と SSHFS を用いた簡単にアドホックな分散ファイルシステムの構築

HandyFS: A simple ad-hoc distributed file system only using FUSE and SSHFS

頓 楠 † 田浦 健次朗 †† 米澤 明憲 †

Nan DUN Kenjiro TAURA Akinori YONEZAWA

†,†† 東京大学大学院情報理工学系研究科

Graduate School of Information Science and Technology, The University of Tokyo

†{dunnan,yonezawa}@yl.is.s.u-tokyo.ac.jp ††tau@logos.t.u-tokyo.ac.jp

HandyFS provides a simple and efficient interface to enable user ad-hoc merge many remote directory trees (source directory) on distributed machines into one single directory hierarchy (target directory). All operations on the target directory are mapped to corresponding individual source directory. Source directories can be add/remove dynamically at runtime and these changes are reflected to target directory.

HandyFS is implemented only using FUSE module and SSHFS and fully functions in userspace. It merely needs simple configuration on client-side and utilizes standard SSH daemon as servers. By using SSH connection, HandyFS can also easily work under typical network configurations (e.g. NAT, Firewall) to adapt to the wide-area Grid environments.

1 Introduction

Data file sharing is a basic problem in the field of distributed and parallel computing. One conventional solution to this problem is to build a distributed file system. There are many aspects concerning a distributed file system: data availability, usability, scalability, simplicity and performance, etc. Many distributed file systems have been developed to approach all of these goals, such as NFS [7], AFS [2], PVFS [9], and Gfarm [5].

Among these objectives, we focus on offering HandyFS, a simple ad-hoc distributed file system with high usability. HandyFS provides a simple and efficient interface to enable user ad-hoc merge many remote directory trees (source directory) on distributed machines into one single directory hierarchy (target directory). All operations on the target directory are mapped to corresponding individual source directory. Source directories can be add/remove dynamically at runtime and these changes are reflected to target directory.

HandyFS is implemented only using FUSE module and SSHFS and fully functions in userspace. Combined with SSHFS, it merely needs simple configuration on client-side and utilizes standard SSH daemon as servers. By using SSH connection, HandyFS can also easily works under typical network configurations (e.g. NAT, Firewall) to adapt to wide-area Grid environments.

2 Related Works

2.1 FUSE

FUSE (Filesystem in Userspace) [3] is a framework that enables general users to build their own file systems in userspace without acquiring special privilege or modifying OS's kernel source code.

FUSE is implemented as two main components. One of them is an OS kernel module, the other is a userspace library. FUSE functions by creating and managing a translation/mapping between file system calls hooked by kernel module in kernel space and user-specified operations in user space. To achieve this translation, a userspace utility (`fusermount`) is provided to let user specify the mount point while the kernel module creates a special character device (`/dev/fuse`) via which the arguments and return values are passed from kernel to userspace, and vice versa.

2.2 SSHFS

SSHFS (SSH File System) [10] is a remote file system that is implemented by using the SSH protocol over the FUSE.

The idea of SSHFS is to map file system calls to SSH 's sftp client request calls, such that the file access requests to the mount point can be redirected to corresponding SSH requests to remote server. By this approach, users can access data files on remote servers in the same way as accessing local files.

A typical SSHFS usage procedure is shown in Figure 1. User creates a mount point for SSHFS

```
1: user@localhost:~$ mkdir /rhost_mnt
2: user@localhost:~$ sshfs rhost.net:   \
/rhost_mnt
3: user@localhost:~$ [ls rm mkdir] /rhost_mnt
4: user@localhost:~$ fusermount -u /rhost_mnt
```

Fig. 1: Basic Usage of SSHFS

(line 1). User mounts a remote server to local mount point. Access remote server as local directory (line 3) and unmount SSHFS (line 4).

SSHFS has several advantages over conventional remote file systems in terms of usability and simplicity. First, since SSHFS uses standard SSH protocol to talk to remote sftp daemon server, users only need to specify the address and directory of remote server and mount them to local mount point at client side without server side configuration. Second, SSHFS can easily get through typical network configuration in the wide-area grid environments, such as firewall and NAT. Finally, SSHFS inherits its encryption mechanism from SSH such that it is able to perform secure file transfer over untrusted networks.

Although SSHFS is easy to be used to share data files on remote server, it has a limitation that users have to create one mount point for each remote server. A typical one is that simply using SSHFS cannot construct a global file namespace for many remote servers to be shared.

2.3 UnionFS and FunionFS

UnionFS (A Stackable Unification File System) [1] is a file system that allows users to merge multiple directories (branches) into one single overlaid virtual file system. The different branches can be merged as read-only or read-write file systems, so that writes to the virtual file system can be directed to underlying specific real file system. UnionFS allows branch manipulations (e.g., add, remove change mode) at run-time and the priority of branches to be specified to solve the problem when two branches have the same filename. UnionFS is mainly used in Live-CD or diskless file system, as well as sandboxing, snapshot taking, and file servers unifying.

UnionFS is implemented as a kernel module. User can use UnionFS utilities to control the behavior of kernel module. However, non-privileged users cannot install or configure UnionFS.

To overcome the privilege restriction of UnionFS. FunionFS [4] is developed in user space by using FUSE. It aims to have the same semantic as UnionFS does. The main purpose of FunionFS is also to provide a utility to manufacture Live-CD. The implementaion of FunionFS shows that it use in-memory cache to record underlying branches, and a merged hierarchy view is contructed at run-time by asking branches' content. When the number of branches is small, it is efficient. But this implementation will not scale when a large number of branches is involved. Besides this, FunionFS have to re-construct merged file system at every mount time since it does not store any previous merge information.

2.4 Gfarm

Gfarm (Grid Datafarm) [5] is a distributed file system aiming to provide large-scale and high-performance data sharing service over networks.

Gfarm mainly consists of two kinds of nodes: Storage nodes and metadata server node. The Gfarm storage nodes provide basic resources, such as physical data storage and CPU time, to be shared by the whole file system. The Gfarm metadata

server (central server) node is in charge of collecting and managing metadata of data files distributed over many storage nodes. The Gfarm daemon (`gfsd`) and meta-server daemon (`gfmd`) are running on storage nodes and metadata server, respectively. To use Gfarm, both `gfsd` and `gfmd` should be properly installed and configured in advance by privileged users.

3 Design

3.1 Basic Usage

The usage of HandyFS is as simple as SSHFS, as shown in Figure 2.

```
1: user@localhost:~$ mkdir /rhosts_mnt
2: user@localhost:~$ handyfs rhost1.net:  \
rhost2.net:  rhost3.net:  /rhosts_mnt
3: user@localhost:~$ [ls rm mkdir] /rhosts_mnt
4: user@localhost:~$ fusermount -u /rhosts_mnt
```

Fig. 2: Basic Usage of HandyFS

First, user can create an empty directory as mount point for HandyFS (line 1). Then user can mount as many as remote hosts to current mount point (line 2). Finally, user can operate on mount points like local one (line 3) or unmount HandyFS (line 4). By these commands, user can easilty created a merged virtual namespace from specified source directories on multiple remote servers.

3.2 Framework Overview

The framework of HandyFS is shown in Figure 3.

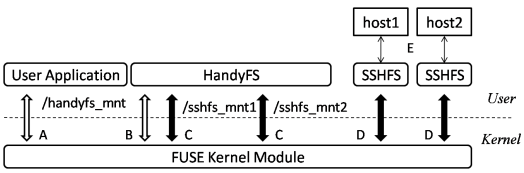

Fig. 3: Framework of HandyFS

The main idea of HandyFS is to use FUSE twice. First, mount points for are created and mounted by SSHFS (e.g., `/sshfs_mnt1` and `/sshfs_mnt2`), such that applications can access data files via mount points (e.g.,data flow C, D and E) in remote servers (e.g., host1 and host2). Secondly, a mount point (e.g., `/handyfs_mnt`) for HandyFS is created and mounted by HandyFS. Then user application can access the virtual file system (e.g., data flow A and B) via target directory (i.e., `/handyfs_mnt`) constructed from source directories (i.e., `/sshfs_mnt1` and `/sshfs_mnt2`) by HandyFS, and the target directory plays as the root directory, or " / ", of the merged file system.

3.3 File System Operations

3.3.1 Directory Merge

Directories merge is the fundamental operations in HandyFS. A directories merge process can also be stated as the answer to following question: "Given a request file in merged virtual file system, in which source directory does it exist?" A naive approach is to ask every source directory whether it holds request directory every time when a query comes. In this case, if there are N source directories, then we have to send N of requests to underlying source directories, which correspond to messages sent by SSHFS to remote servers.

HandyFS use a directory table to record the mapping from the directory of virtual file system to the directories. Suppose there are two source directories, `/srcdirA` and `/srcdirB`, and they both include a sub-directory with the same name cmndir (i.e., `/srcdirA/cmndir` and `/srcdirB/cmndir`), then there will be an entry in directory table as Figure 4 shows.

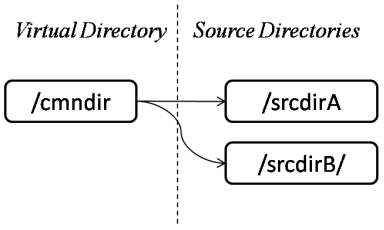

Fig. 4: Diretory Table

Therefore, when a request comes to access `/cmndir` in virtual directory, HandyFS will direct this request to `/srcdirA/cmndir` and `/srcdirB/cmndir`, according to the directory

table. Thus the number of messages is heavily reduced by limiting the request to be delivered to source directories that holds required directory.

At the beginning of file system mount, only the root directory of the virtual file system will searched and added to the directory table. For other subdirectory, HandyFS lazily leaves their appending until actual file requests reach them and then build them on demand.

3.3.2 Directory Split

Directories split is the reverse operation of directories merge. Based on the description in Section 3.3.1, when a source directory is split off from merged virtual directory, it suggests that all directory table entries merged from this source directory should be removed. So the next time virtual file system will not dispatch requests to previously removed source directory. However, this operation is costly and will slow down the whole performance since all directory table entries should be traversed and checked, whether they includes removed source directory or not.

To make this manipulation efficient, an on-demand approach is employed again by using filtering. A filter is used to remember current merged source directories. When one of them is removed, it will be immediately forgotten by the filter. Then even entries having this source directory is retrieved from directory table during query, they will be filtered out by filter, such that merged file system will be blind to this removed source directory. Finally, entries having removed directories in directory table will be removed later by garbage collector in the background.

3.3.3 File Lookup

HandyFS only remember directory merge (i.e., directory table) and never holds any information of non-directory files. This policy greatly simplified metadata management in HandyFS.

Therefore, to lookup files, HandyFS first parse file ŝ full pathname to get its parent directory. Then HandyFS checks whether the pathname of parent directory is in directory table. If it exists in directory table, then HandyFS will retrieve corresponding source directory entries and search in those real source directories for target files.

Otherwise, non-existence of this parent directory in directory suggests that this directory may not exist or have not been added to table directory. Then the parent directory of current looking up parent directory is parsed and used to perform the same lookup. HandyFS recursively searches upward until it reaches root directory or find a source directory having requested pathname.

3.3.4 File Creation

File creation can be creating a directory file or non-directory file. To create a non-directory file, HandyFS first check the availability of target file 's parent directory. If the parent directory does exist, it will create a file under it. However, a target parent directory may be merged from several underlying source directory. Current design lets HandyFS randomly choose a source directory to perform real write operations.

Creation of a directory is mostly the same as above procedure, except that the newly created directory should be registered into directory table for further looking up.

3.3.5 Directory Read

HandyFS reads a directory by first perform a looking up of target directory as mentioned in Section 3.3.1. Then the source directory entries for this target directory are obtained from directory table. Finally, `readdir()` operation are executed on each source directory, and all directory entries and gathered and returned to users.

3.4 Existing Problems

Operations in Section 3.3 are not complex. However, to implement HandyFS and put it into practice usage will face several problems.

The first problem arises when multiple source directories contain files with the same filename. To comply with UNIX file system semantic, duplicate

files are not allowed to return to users, as well as files with same filename may have different attributes. This problem is also described in [11] and a solution is to define priority of source directories, so that the virtual file system will always return the file residing in directory with high priority to user.

Another problem is about the stale of cache that contains sourece directory information. Since source directories may change dynamically, these changes should be refected to the merged virtual file system. HandyFS should be aware of this situation. One possible solution is to define a expiration time, such that HandyFS will update its directory table when it feels that the cache may be out-of-date.

4 Implementation

The implementation of HandyFS consists of about 2,000 lines of C code. Most of codes are to implement FUSE library interface. The rest is devoted directory merge operations.

BerkeleyDB [8] provides a simple and high efficient interface to achieve store/retrieve manipulation on key/data pairs. BerkeleyDB also provides native support for duplicate data and multithreading. BerkeleyDB's key and data naturally corresponds virtual directory and source directories in directory, respectively. Thus, the store and management of directory table is implemented by BerkeleyDB.

The filter described in Section 3.3.2 is implemented by a in-memory bucket-like data structure. Each source directory is hashed and stored into the proper bucket. By this means, the filter is able to quickly locate target source directory hash value to perform add/remove operations.

5 Experiments and Evaluation

5.1 Experimental Environments

HandyFS was evaluated by two experiments performed on a wide-area gird platform called InTrigger [6]. The InTrigger platform currently consists of six sites located all over Japan. They are "hongo", "chiba", "suzuk", "okubo", "imade", and "kyoto". Homogeneous nodes of first five sites are chosen out to run these experiments. The specification of these nodes is shown in Table 1.

Table 1: Specification of Experiemental Machines

Hardware/Software	Specification
CPU	Core2 Duo 2.13GHz
Memory	4GB
Network	Gigabit Ethernet
OS	Linux 2.6.18
FUSE Kernel Interface	7.8
FUSE Library	2.7.0
SSHFS	1.8

5.2 Evaluation

Two experiments are carried out to evaluate the performance of HandyFS.

The first experiments is to measure data transfer rate. A 512MB data file is transferred from *hongo* site to other four sites. These data transfers are performed using different method: ftp, sftp, sshfs and handyfs.

The second experiment is to test the time of compiling OpenSSH source code as a benchmark of file system operations. These compilations were executed on local file system, FUSE mounted local file system, NFS, SSHFS and HandyFS. Here the NFS is used within a cluster, while SSHFS and HandyFS are used among clusters.

5.2.1 Experimental Results

The results of two experiments described above are shown in Figure 5 and Figure 6, respectively.

From Figure 5, it can be read that underlying network bandwidths are different from each pair of sites. The bandwidths of *hongo-chiba* and *hongo-suzuk* are much higher than that of *hongo-okubo* and *hongo-imade*. To transfer the same mount of data, Ftp can utilize higher bandwidths to achieve faster data tranfer rate. On the other side, sftp is less sensitive to the network bandwidths. Inherently, SSHFS and HandyFS, which both employ sftp protocol, are also slow in data transfer rate. On average, HandyFS only achieved about 30% performance of sftp, or 50% performance of SSHFS.

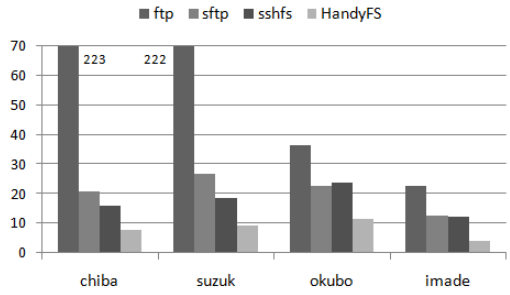

Fig. 5: Data Transfer Rate (Mbps)

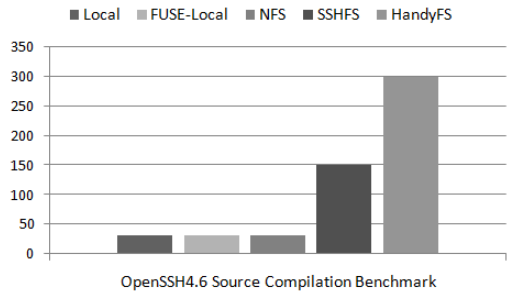

Fig. 6: Source Compiling Benchmark (secs)

In Figure 6, SSHFS and HandyFS cost much more time than other three file system. This is reasonable becasue SSHFS and HandyFS run in a relative higher latency environments. In this experiment, HandyFS's execution time is about as twice as much as SSHFS's execution time.

5.2.2 Results Analysis

The analysis of experimental results is critical. Because it not only shows the direction of further performance enhancement, but also leads a better understanding of the behaviors of other building blocks (e.g., sftp protocol and FUSE module).

Results in Section 5.2.1 demonstrate that sftp protocol becomes a bottleneck for applications built on it. The sftp protocol will encrypt data before transferring them over the networks. Therefore, an encryption/decryption overhead is proposed to use certain amounts of CPU. Especially in InTrigger gigabits networking, this extra procedure will significantly slow down the transfer rate.

Further, considerable overhead appears when SSHFS and HandyFS are compared. HandyFS always exhibits a "doubled" execution time over SSHFS. Since HandyFS uses FUSE again above SSHFS, this overhead is supposed to be caused by FUSE. To verify this, another independent experiments was performed to examine the overhead of overlapped FUSE mount. When two or more FUSE mounts is performed on a single local system, because all of mount points talks via kernel module, it is likely that those overlapped/concurrent accesses to mount points will lead underlying FUSE kernel module into heavy load or even race condition. Thus, in terms of this fact, FUSE limited the scalablity of HandyFS.

6 Conclusions and Future Works

By using FUSE and SSHFS, HandyFS — a simple ad-hoc distributed file system — has been designed and implemented. Although with performance and scalability limitations, it still exhibits a way of building easy-to-use file sharing utility.

Future works includes solving the problems described in Section 3.4 and a deep investigation of FUSE runtime mechanism to enhance performance and scalability of HandyFS.

Acknowledgements

This work is part of the research project "Cyber Infrastructure for the Information-explosion Era", which is sponsored by MEXT Grant-in-Aid for Scientific Research on Priority Areas.

References

[1] A Stackable Unification File System: `http://unionfs.filesystems.org`.

[2] Andrew File System: `http://www.openafs.org`.

[3] Filesystem in Userspace: `http://fuse.sourceforge.net`.

[4] Funion File System: `http://funionfs.apiou.org`.

[5] Grid Datafarm: `http://www.gfarm.org`.

[6] InTrigger Platform: `https://www.logos.ic.i.u-tokyo.ac.jp/intrigger/`.

[7] Network File System:
http://nfs.sourceforge.net.

[8] Oracle BerkeleyDB:
http://www.oracle.com.

[9] Parallel Virtual File System:
http://www.pvfs.org.

[10] SSH File System:
http://fuse.sourceforge.net/sshfs.html.

[11] Wright, C. P., Dave, J., Gupta, P., Krishnan, H., Quigley, D. P., Zadok, E., and Zubair, M. N.: Versatility and Unix Semantics in Namespace Unification, *ACM Transactions on Storage*, Vol. 1,No. 4(2005), pp. 1–29.